Chapitre 1

Visualisations pour la remodularisation à large échelle des systèmes à objets

Dans ce chapitre, nous abordons deux points critiques de la remodularisation : comment visualiser la structure d'un logiciel pour aider à la rendre modulaire et comment aider le développeur à prendre les bonnes décisions. D'abord nous décrivons certains outils pour visualiser la structure des logiciels et comment nous les adaptons à la remodularisation. Ensuite nous présentons des visualisations adaptées à l'identification des problèmes de modularité. Enfin, nous proposons un outil, nommé Orion, permettant de simuler les changements de structure d'un logiciel. Il permet d'analyser l'impact des changements dans la structure et d'évaluer les coûts associés.

1.1. De la remodularisation

La remodularisation est l'ensemble des techniques permettant de rendre un système modulaire. L'objectif est de rendre un logiciel flexible, configurable et évolutif [PAR 72]. La question de la modularité se pose aussi pour les systèmes à objets placés au cœur des entreprises : des systèmes critiques souvent de taille conséquente ayant évolués au fil des années [DEM 02].

Un développeur s'occupant de tels systèmes doit remplir quatre objectifs : il doit (1) identifier les problèmes, (2) résoudre ces problèmes, (3) éviter la régression du système et (4) minimiser les coûts de changement [BOH 96].

Chapitre rédigé par Jannik LAVAL et Simon DENIER et Stéphane DUCASSE.

Le point (1) nécessite l'utilisation d'outils adaptés à la tâche à réaliser. Ils permettent au développeur de cibler rapidement les parties critiques du système. Dans le cadre de l'ingénierie, des rapports de bug ou des tests assurent cette fonction. Dans le cadre de la remodularisation, l'utilisation de visualisations ou de métriques est plus appropriée. Nous discuterons plus en détail de ces outils dans les sections 1.2 et 1.3. Les points (2) et (3) sont assurés principalement par les compétences du développeur et par les tests du système.

Les points (3) et (4) soulèvent une question : comment s'assurer qu'un changement dans la structure du logiciel est à la fois bénéfique et rentable ? Bien souvent, faute d'outils adaptés, le développeur juge cela en fonction de son expérience et de son intuition.

Ceci soulève donc les questions suivantes : comment visualiser la structure d'un logiciel pour aider à la rendre modulaire ? Comment aider le développeur à prendre les bonnes décisions. Dans ce chapitre, nous donnons des pistes de réponse à ces deux questions en décrivant des outils pour visualiser la structure des logiciels et comment nous les adaptons à la remodularisation. Ensuite nous présentons des visualisations adaptées à l'identification des problèmes de modularité. Enfin, nous proposons un outil, nommé Orion, permettant de simuler les changements de structure d'un logiciel. Il permet d'analyser l'impact des changements dans la structure et d'évaluer les coûts associés. Ainsi, le développeur peut décider d'un plan de changements de son systèmes.

Les travaux présentés dans ce chapitre sont pour la plupart implémentés au-dessus d'une plate-forme d'analyse de logiciels, appelée Moose [NIE 05]. Elle fournit un ensemble d'outils pour la création, l'analyse et la visualisation de modèles FAMIX[1] [DUC 09a] pouvant s'adapter aux besoins de chaque représentation (statique, dynamique, historique, etc.).

1.2. Défis pour la remodularisation des systèmes patrimoniaux

1.2.1. *Problèmes*

Les grandes applications développées avec des langages à objets évoluent souvent durant plusieurs années avec plusieurs changements d'équipes de développement. Les deux facteurs, âge de l'application et *turn-over* des équipes de développement, impliquent une perte de connaissance non négligeable et un développement souvent moins fluide : duplication de code, dépendances indésirables entre entités, architecture de moins en moins modulaire [DEM 02, MAR 03]. Il n'est pas rare qu'une application de taille moyenne soit composée de plusieurs centaines de paquetages. C'est

1. Voir [DEM 01] et www.moosetechnology.org/docs/famix.

pourquoi comprendre comment sont formés les paquetages, leur organisation interne et leurs communications externes dans une application est un défi ayant pour objectif la compréhension de la structure applicative globale.

Deux paramètres sont à considérer dans ces systèmes pour évaluer la complexité de la remodularisation [ABD 09b]. Le premier est la taille de l'application. Plus une application est grande, plus les problèmes de structure sont nombreux et moins ils sont visibles. Il devient difficile d'identifier les sources de problèmes. Le deuxième paramètre est la remédiation. Lors de modifications de ces logiciels, il est difficile de prévoir l'impact sur les fonctionnalités et la structure du logiciel. Ainsi, la remodularisation se fait en itérant de petites modifications suivies de vérifications.

Il est donc important de comprendre l'organisation des paquetages et de leurs relations. Les paquetages sont des structures complexes car elles contiennent des classes et des méthodes en relation entre elles et avec l'extérieur. Il est donc important de comprendre quel rôle peut jouer un paquetage dans le système (paquetage noyau, paquetage de fonctionnalités...). Les développeurs doivent donc comprendre comment les paquetages sont structurés et quelles sont leurs relations avec les paquetages voisins.

En quelques questions, nous présentons les informations utiles à la compréhension des paquetages et à leur remodularisation. Cette liste n'est pas exhaustive, elle montre seulement les problèmes qu'un développeur peut rencontrer [ABD 09b].

– taille : quelle est la taille d'un paquetage en termes de classes, de relations d'héritage, de références internes et externes, des clients ? Quelle place tient un paquetage dans le système ?

– cohésion et couplage : l'évolution d'un paquetage aura des impacts sur le système selon son couplage et sa cohésion [ABD 11, ARI 04, BRI 99]. Comment prévoir l'impact d'un changement d'un paquetage sur le système ?

– paquetage central *versus* périphérique : est-ce qu'un paquetage appartient au noyau de l'application ou aux modules périphériques ? Est-ce qu'un paquetage est fournisseur ou utilisateur d'un service [ABD 08] ?

– un développeur ou une équipe : qui a développé le paquetage ? Qui le maintient ? Connaître ces personnes ou ces équipes aide à comprendre l'architecture et le rôle des paquetages [Gˆ 05a, POL 07].

Un bon paquetage devrait être autocontenu, ou avoir seulement quelques dépendances à d'autres paquetages clairement identifiées [ABD 09a, D'A 06, DUC 07, HAU 02]. Un paquetage peut interagir avec d'autres en tant que fournisseur, client ou les deux [ABD 08]. Il peut faire beaucoup de références aux autres paquetages ou seulement à quelques-unes. Si un paquetage définit des sous-classes d'un autre paquetage, on pourrait le considérer comme un sous-paquetage. Martin définit des lignes de

conduite pour la conception des paquetages [MAR 03], en particulier basées sur l'idée qu'un paquetage est une unité de réutilisation.

La compréhension d'un paquetage est difficile et les outils de visualisations ont pour objectif de réduire cette difficulté [ABD 08, BER 83, DUC 07, TUF 01, WAR 00].

1.2.2. *Approches de visualisation*

Deux grands types d'approches permettent de fournir des informations sur les paquetages et leurs relations : les métriques et les visualisations [ABD 10, LAN 03a]. Les métriques permettent de mesurer la qualité du logiciel et de manipuler des chiffres tandis que les visualisations mettent en évidence certains aspects du code par l'utilisation de différentes couleurs et formes [KAR 03]. Les deux approches sont complémentaires dans le sens où la visualisation est particulièrement adaptée à la représentation de métriques. Ce paragraphe traite particulièrement des visualisations existantes mélant utilisant des métriques car elles offrent un moyen rapide d'analyser l'état d'un logiciel [LAN 03b].

1.2.2.1. *Graphes orientés*

– principe : la visualisation de graphes orientés se fait souvent à l'aide de liens entre deux nœuds caractérisant une dépendance. Plusieurs outils (dont GraphViz[2], Walrus[3] ou Guess[4]) peuvent être utilisés pour générer ce type de visualisation.

– avantages : elle est la plus intuitive des visualisations. Elle permet de voir en quelques instants la complexité du logiciel, par l'analyse du nombre de liens entre les nœuds.

– inconvénients : la mise en forme n'est pas adaptée aux systèmes dont le nombre d'éléments visualisés dépasse les quelques dizaines [HEN 07]. Cette visualisation est donc peu adaptée aux systèmes à objets en particulier si l'on souhaite des détails sur l'intérieur des paquetages.

1.2.2.2. *TreeMap*

– principe : une TreeMap [BAL 05, JOH 91] présente des informations hiérarchiques sous une forme compacte pour maximiser l'utilisation de l'écran. La hiérarchie complète est représentée par un rectangle (figure 1.2). Chaque branche est représentée par un rectangle contenu dans le rectangle parent. Le premier rectangle est coupé verticalement. Les rectangles du niveau du dessous sont coupés horizontalement, le suivant, de nouveau verticalement, et ainsi de suite. Chaque découpe est

2. www.graphviz.org/.

3. www.caida.org/tools/visualization/walrus/.

4. graphexploration.cond.org/.

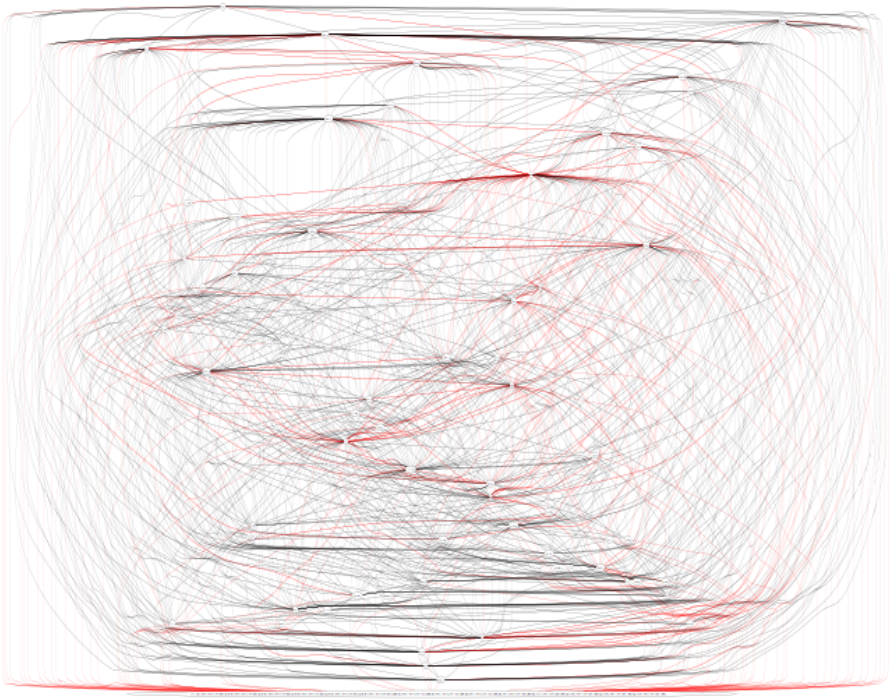

Figure 1.1 – Relation entre les paquetages du *framework* Pharo-Core 1.0 - 70 paquetages

définie proportionnellement à la taille de chaque sous-système. La figure 1.2 illustre le principe de TreeMap et compare un arbre et une TreeMap. Cette dernière est plus compacte.

– avantages : dans le cadre de la remodularisation, cette visualisation permet de voir rapidement les paquetages (ou les classes) du système les plus importants en taille ou les hiérarchies de classes les plus complexes.

– inconvénients : cette visualisation ne permet pas de bien comprendre les liens complexes entre les paquetages. Même si elle permet de visualiser correctement des structures en arbre, ce n'est pas le cas pour les structures plus complexes. Ainsi, il faut noter que la remodularisation d'applications passe nécessairement par des informations plus détaillées.

1.2.2.3. *Verso*

– principe : Verso [LAN 05] est un *framework* générique pour visualiser

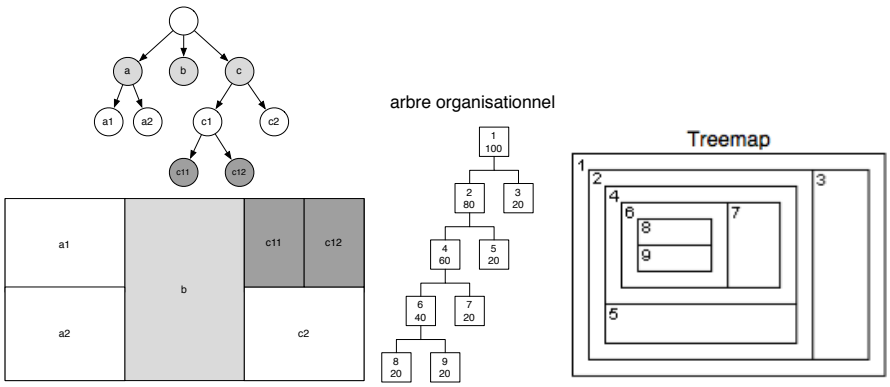

Figure 1.2 – Gauche : une TreeMap montrant six nœuds répartis sur quatre niveaux hierarchiques ; droite ; comparer une TreeMap et un arbre organisationnel (pris dans l'article [BAR 01]).

l'organisation de logiciels et leurs métriques. Il utilise un espace en 2D dans lequel il place des éléments en 3D pour faciliter la détection de propriétés. L'utilisateur peut donc naviguer dans la visualisation pour détecter des éléments intéressants et se focaliser dessus. Les entités (par exemple des classes) sont représentées par des boîtes en 3D, et les métriques par les attributs des boîtes (hauteur, angle, couleur). Verso utilise une organisation des éléments basée sur des régions comme le fait TreeMap. La figure 1.3 montre la localisation des classes dans une organisation hiérarchique de paquetages. Chaque classe est représentée par une boîte avec trois métriques appliquées : CBO (*Coupling between Objects*, nombre d'autres classes auxquelles une classe est couplée), LCOM5 (*Lack of Cohesion of Methods*, une mesure de la cohésion d'une classe) et WMC (*Weighted Methods per Class*, somme de la complexité cyclomatique de McCabe) qui sont respectivement représentées par la couleur, l'angle et la hauteur. Dans cette figure, on voit les boîtes verticales (nord-sud) ont un résultat de LCOM5 faible contrairement aux boîtes horizontales ; et les petites boîtes ont une valeur de WMC faible.

– avantages : ce type d'associations de métriques à des propriétés graphiques permet de voir très rapidement les classes complexes dans le système : elles sont rouges, hautes et fortement orientées vers l'horizontal.

– inconvénients : cette visualisation permet de voir la structure de chaque entité étudiée. Cependant, elle est difficilement utilisable dans le cadre de la compréhension des relations entre les paquetages, où le développeur a besoin de connaître la structure mais aussi les communications.

CodeCity reprend la métaphore de la ville ainsi que son évolution au cours du temps pour montrer certaines propriétés des grands logiciels [WET 07, WET 08].

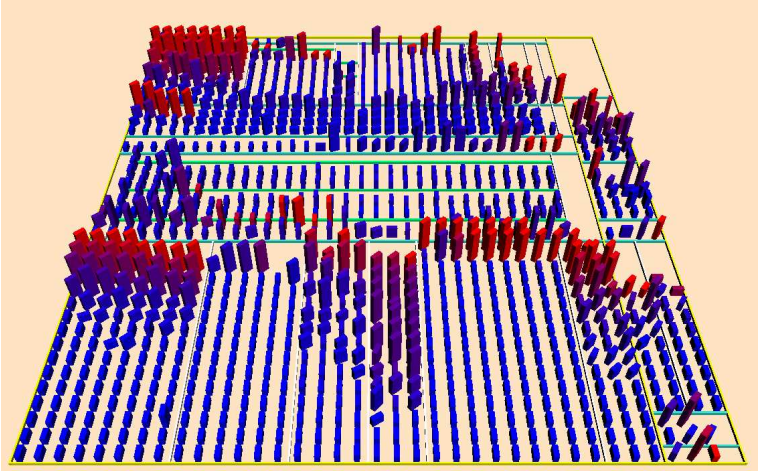

Figure 1.3 – Visualisation de classes organisées par paquetage

1.2.2.4. *Polymetric Views [DEM 99, Gˆ 05b, LAN 03b]*

– principe : l'idée est d'offrir une visualisation légère où la représentation graphique d'une entité (paquetage, classe, méthode) est enrichie avec des métriques. La visualisation utilise un affichage en deux dimensions. Un nœud représente une entité et un lien représente une relation entre deux nœuds. Cette représentation simple est enrichie par l'ajout de cinq paramètres pour visualiser des métriques sur chaque nœud (montré dans la figure 1.4 (gauche)) et de deux paramètres sur les liens. Un nœud a donc cinq attributs définis sur sa longueur, sa hauteur, sa couleur, sa position horizontale et sa position verticale. Un lien a deux attributs : son épaisseur et sa couleur. Chacun de ces attributs peut recevoir le résultat d'une métrique. La figure 1.4 (droite) montre un SystemComplexity, qui, suivant une Polymetric View, met en évidence l'arbre d'héritage et mesure la taille des classes (nombre de méthodes (NOM), nombre d'attributs (NOA), nombre de lignes de code (WLOC)).

– avantages : cette visualisation permet de voir très rapidement (tout comme Verso) les éléments complexes grâce à leur taille et leur couleur. Elle est également intuitive car elle se rapproche des graphes par l'utilisation de boîtes et de liens.

– inconvénients : Polymetric Views dépend de la structure de l'application : si l'application est bien structurée, la visualisation est plutôt d'une bonne qualité, sinon, les liens entre les entités font perdre du sens à la visualisation.

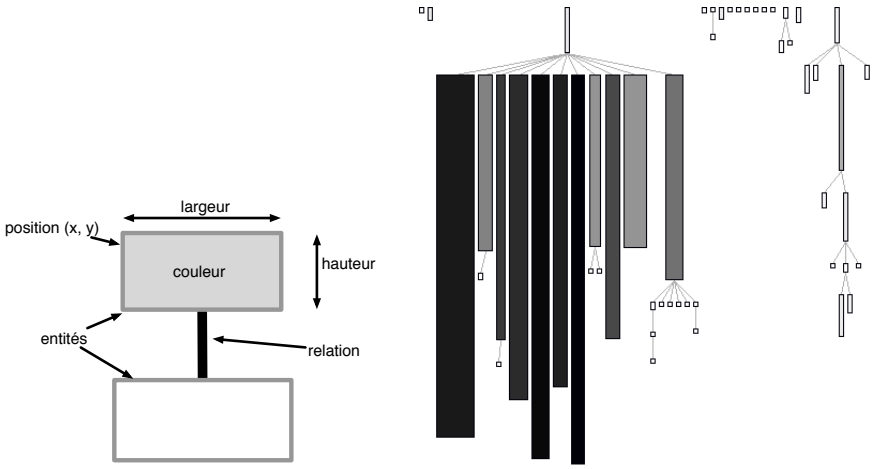

Figure 1.4 – Gauche : principe de Polymetric Views ; droite : System complexity ; largeur : NOA. hauteur : NOM, couleur : WLOC

1.2.2.5. *Package Fingerprint*

– principe : Package Fingerprint [ABD 10] est visualisation des références entrantes et sortantes d'un paquetage. Elle est sémantiquement riche, compacte et zoomable. Elle est centrée sur les paquetages. Deux vues sont proposées : références entrantes et sortantes. Nous ne présentons ici que la forme compacte de la visualisation pour des raisons de place. La figure 1.5 illustre le principe de Fingerprint. Le coin en haut à gauche indique le paquetage en cours d'évaluation. Les bords représentent les paquetages qui référencent le paquetage analysé. L'ordre de gauche à droite et haut en bas est le nombre de classes référençantes. La figure 1.5 montre les trois paquetages qui référencent P_1 : P_3, P_4 et P_2, qui référencent respectivemment quatre, trois et deux classes de P_1. Les cellules de la diagonale présentent le nombre de classes auquel le paquetage fait référence. Les autres cellules présentent l'intersection des classes utilisées simultanément par les deux paquetages. La figure 1.6 présente le Fingerprint entrant du paquetage *Network : :SqueakPage* du *Squeak38*. Elle montre que les paquetages *Morphic : :Kernel* et *Morphic : :Books* sont les paquetages référenceurs dominants. Ils référencent toutes les classes de *Network : :SqueakPage* (ici seulement trois classes : *SqueakPageCache*, *SqueakPage* et *URLMorph*). Le reste des paquetages référencent différents groupes de ces trois classes : les paquetages *Kernel : :Objects* et *Morphic : :Worlds* référencent la classe *SqueakPageCache* (*Service1*) ; les paquetages *System : :Support* et *Tools : :FileList* référencent la classe *SqueakPage* (*Service2*) ; le paquetage *Morphic : :Widgets* référence, en plus de *Service2*, la classe *URLMorph*.

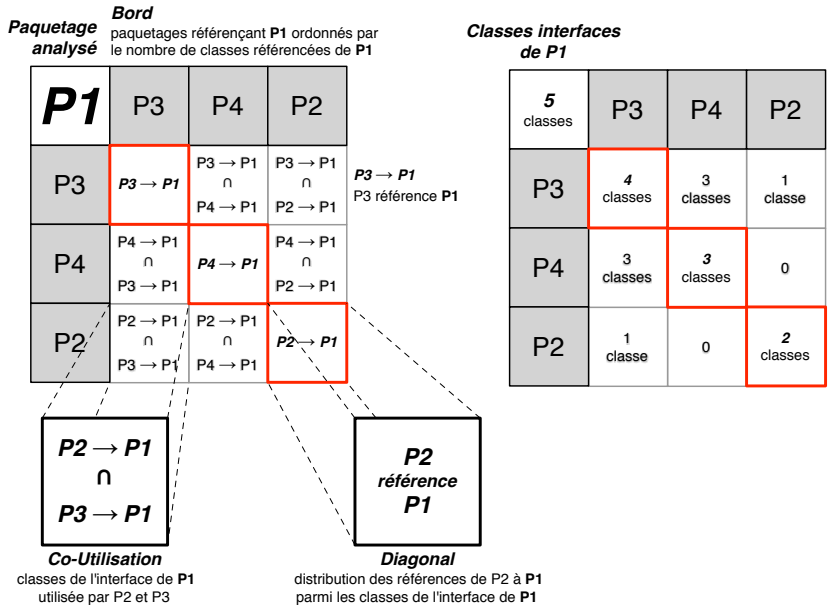

Figure 1.5 – Structure d'un Fingerprint entrant pour le paquetage P_1.

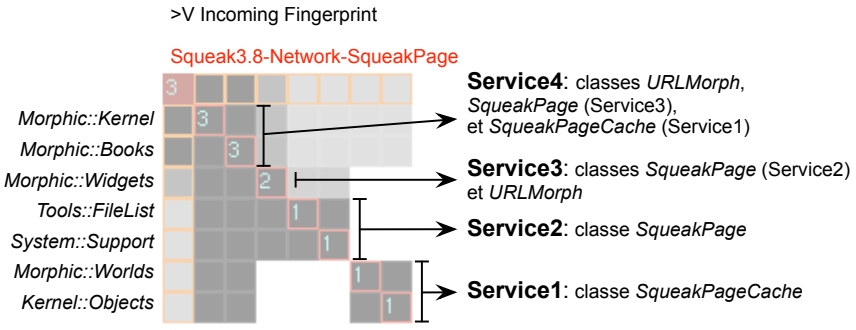

Figure 1.6 – Fingerprint du paquetage *SqueakPage* du sous-systeme*Squeak38 : :Network*

– avantages : Package Fingerpint propose une visualisation permettant de comprendre comment un paquetage utilise et est utilisé par de multiples clients simultanément. Il permet de voir la co-utilisation des classes que cela soit en tant qu'utilisatrices

ou qu'utilisées. La visualisation est compacte et centrée sur un paquetage ce qui permet de se faire une idée de l'utilisation d'un paquetage rapidement.

– inconvénients : Package Fingerprint est parfois trop dense et il est difficile de comprendre immédiatement toutes les subtilités de la visualisation. En mode zoom arrière les visualisations sont plus simples. Cette visualisation ne montrent pas les dépendances entre paquetages ou les possibles cycles.

1.2.2.6. *DistributionMap*

– principe : DistributionMap [DUC 06] est une visualisation générique montrant comment des propriétés sont réparties sur des entités. Le principe de DistributionMap est de représenter les entités internes au système étudié (des sous-systèmes, paquetages, classes, etc.) en spécifiant une couleur pour chaque propriété. La figure 1.7 illustre le principe de DistributionMap avec cinq conteneurs (rectangle noir) contenant respectivement 6, 2, 5, 10 et 14 éléments. Ces éléments sont répartis parmi quatre propriétés (rouge, bleu, vert et jaune). Nous pouvons interpréter : la propriété bleue est très localisée, la jaune est transversale, la verte est comme un poulpe avec son corps (part 1) et ses tentacules (part 3 et part 4). On peut aussi voir que les conteneurs part1 et part5 ne contiennent qu'une seule propriété.

– avantages : DistributionMap peut être appliquée à de nombreuses propriétés comme des métriques, des informations sémantiques ou les différents auteurs d'un projet. Dans le cadre de la remodularisation, nous pouvons mettre en avant, par exemple, les classes interfaces (les classes qui communiquent avec d'autres paquetages), les classes internes (celles qui communiquent uniquement avec les classes de leur propre paquetage) ou les classes non communicantes. La figure 1.8 montre le cas de JEdit avec les « Core Classes » (rouge), les « UI Classes » (vert), les « Scripting classes » (bleu). Ce type de visualisation donne un aperçu de la complexité d'un paquetage.

– inconvénients : une entité dans une DistributionMap ne peut prendre qu'une valeur parmi celle des propriétés. Cela limite l'ajout de propriétés sémantiques dans la visualisation. De plus, seule la couleur peut porter une propriété, ce qui limite l'utilisabilité pour la remodularisation.

1.2.2.7. *Package Blueprint*

– principe : Package Blueprint [DUC 07] montre comment un paquetage interagit avec les autres paquetages du système. Elle affiche les classes du paquetage et ses relations avec les autres paquetages. Elle est composée de trois parties. L'une informe des paquetages utilisés (*Outgoing blueprint*), une autre des paquetages clients (*Incoming blueprint*) et une dernière les paquetages en relation d'héritage (*Inheritance blueprint*). Package Blueprint est structuré en surfaces représentant les paquetages à l'intérieur desquels des boîtes représentent les classes (figure 1.9). Le paquetage est représenté par un rectangle noir, les autres rectangles sont les paquetages en relation avec ce dernier.

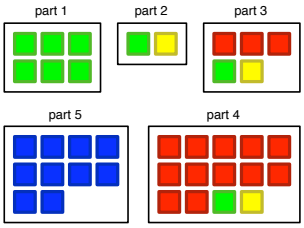

Figure 1.7 – DistributionMap montrant cinq conteneurs et quatre propriétés : rouge, bleu, vert et jaune.

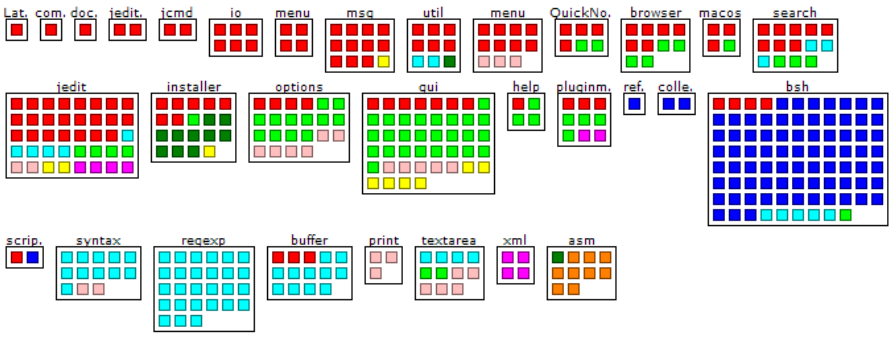

Figure 1.8 – DistributionMap de JEdit montrant les « Core Classes » (rouge), les « UI Classes » (vert), les « Scripting classes » (bleu)

– avantages : elle offre une vision précise du système car elle montre les classes du paquetage ciblé et leurs interactions avec les classes internes et externes au paquetage.

– inconvénients : Package Blueprint rend compte de chaque paquetage pris isolément. Pour une vue générale, il nécessite de naviguer et interagir avec la visualisation.

1.2.2.8. *Dependency Structural Matrix*

– principe : une matrice de dépendance se construit selon le format suivant : une cellule représente le lien entre sa colonne (paquetage source) et sa ligne (paquetage cible) (figure 1.10). Si cette cellule est vide, le lien n'existe pas, si elle contient des informations, le lien existe entre la colonne et la ligne. Les informations contenues dans la cellule peuvent être de différentes formes : une simple croix, une pondération de la dépendance ou toute autre information utile.

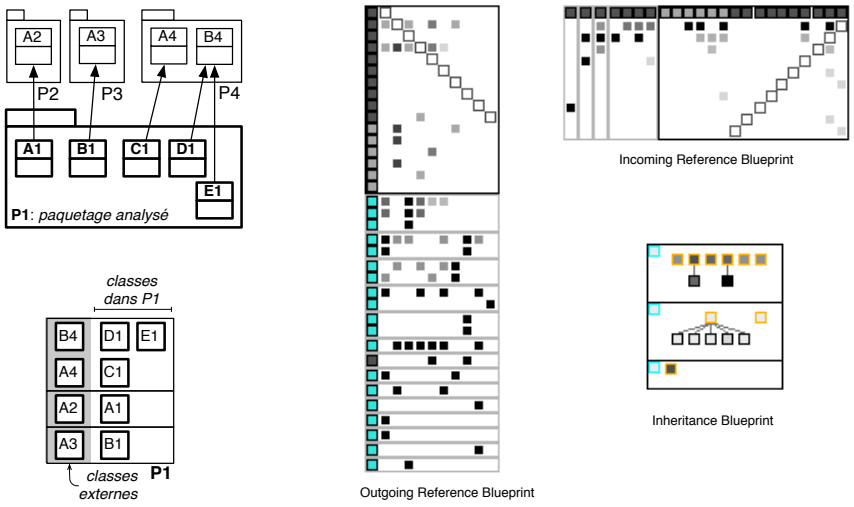

Figure 1.9 – Gauche : principe de Package Blueprint avec quatre paquetages ; droite : les trois visualisations de Package Blueprint mises en œuvre

– avantages : l'utilisation de matrices de dépendance (DSM - *Dependency Structure Matrix*) est une solution éprouvée pour révéler les problèmes de cycles dans une structure [SAN 05]. Elles sont efficaces pour détecter des cycles entre composants logiciels. Les résultats obtenus sont pertinents pour vérifier l'indépendance de différents composants logiciels.

– inconvénients : elles ne fournissent pas suffisamment d'informations fines exploitables pour comprendre la raison d'un cycle. Les matrices de dépendance permettent juste de faire un état des lieux (figure 1.11) des cycles avec une vue à gros grain.

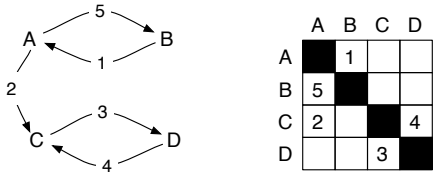

Figure 1.10 – Une matrice de dépendance correspondant au graphe à gauche

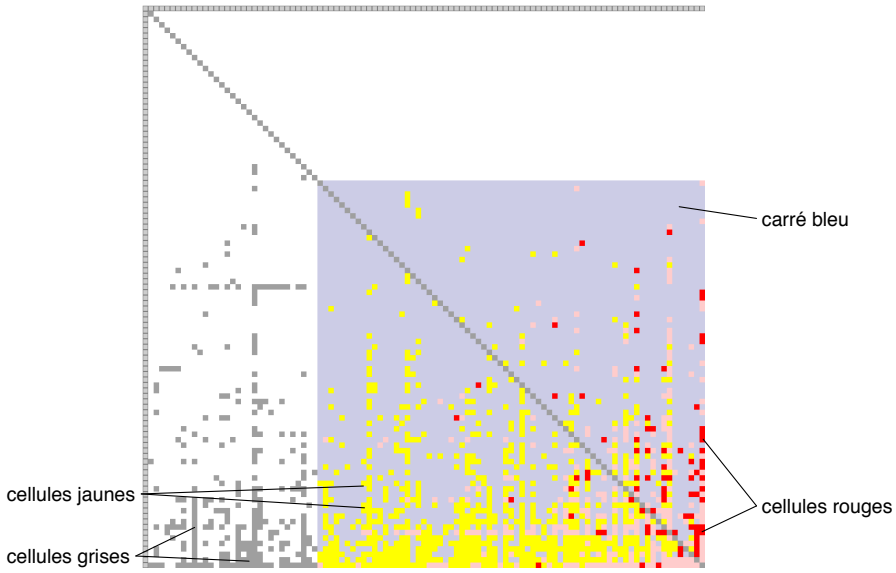

Figure 1.11 – La DSM de Pharo1.0 : le carré bleu représente un ensemble de paquetages en cycle, en rouge les cycles directs, en jaune et en gris les dépendances simples

1.2.3. *Prévoir l'impact et remédier*

L'évolution d'architecture logicielle [DUC 09b], l'analyse de changement d'impact, la mesure de la qualité logicielle, la remodularisation sont des tâches importantes du processus de réingénierie [BOH 96]. Elles demandent aux développeurs de faire des choix importants pour la future architecture du logiciel.

Alors que les développeurs devraient pouvoir tester plusieurs possibilités de remodularisation avant de choisir la plus adaptée, en pratique ils utilisent plus souvent leur expérience et leur intuition. Le problème provient du manque d'outils et de processus adaptés à l'analyse et à la comparaison de plusieurs propositions, pour en choisir la plus appropriée.

S'il était possible de comparer plusieurs évolutions possibles de l'architecture (que l'on nomme ensuite « futurs ») d'un logiciel avant de choisir le plus adapté, une session de réingénierie correspondrait à ce processus. Dans ce cas, le développeur utilise une visualisation pour avoir un aperçu des problèmes et utilise des outils d'analyses métriques pour cibler concrètement le problème. Ensuite, il simule certaines possibilités de changement pour voir quel serait l'impact sur le système. Si un changement n'est pas jugé bénéfique au système, il peut être abandonné sans avoir à modifier le

code source (on est dans un espace de simulation), sinon il valide le changement et l'applique à son code source.

Dans ce scénario, nous avons besoin de :

– naviguer entre plusieurs futurs du même système ;

– appliquer des analyses diverses sur ces futurs potentiels ;

– comparer ces résultats de simulation pour choisir le plus adapté.

Pour cela, il est coûteux de travailler directement sur le code source, nous utilisons donc un modèle de code source sur lequel nous pouvons effectuer les modifications sans impacter le fonctionnement du logiciel.

1.2.3.1. *Naviguer entre les versions*

Naviguer entre les versions d'une même application n'est pas trivial. La majorité des logiciels de contrôle de versions est basée sur des sauvegardes des changements successifs entre les différentes versions. Cela permet d'utiliser un minimum d'espace disque, cependant il ne garantit pas une bonne navigation entre les versions de par le fait qu'une seule version du système est présente dans son ensemble. Il est, bien sûr, possible de générer les versions successives mais cela est coûteux en temps et en place. De plus, ces logiciels gèrent souvent des fichiers, plutôt que des objets. Certaines infrastructures, comme Molhado [NGU 05] permettent de gérer des objets. Les architectures Git (git-scm.com/) et Mercurial (mercurial.selenic.com/) permettent la getion de fichiers ; donc par extension d'objets pouvant être sérialisés et stockés dans des fichiers.

D'autres mécanismes d'enregistrement de changement en temps réel permettent de suivre les changements, comme celui de Smalltalk appelé ChangeSet qui capture la liste des éléments modifiés, que l'on peut ensuite manipuler ou sauvegarder.

Dans tous ces logiciels, la gestion est linéarisée et à tout instant, une seule version est accessible. Changer de version est une manipulation lourde qui rend difficile la navigation et la comparaison des versions en termes de remodularisation, c'est-à-dire de comparer les métriques, d'analyser la structure de plusieurs versions en même temps.

1.2.3.2. *Analyser le changement*

Comparer des versions nécessite une bonne navigation, mais aussi des outils appropriés pour la comparaison. Une approche basique est d'appliquer des métriques ou une visualisation sur plusieurs versions indépendamment et de comparer les résultats. Certains modèles existent pour l'analyse du changement. Par exemple, un modèle basé sur quatre types de liens [CHA 02, HAN 97] entre des entités (association (S), aggrégation (G), héritage (H) et invocation (I)) formule des expressions booléennes pour connaître les impacts du changement. Un changement basique est considéré comme

S H+G. D'autres algorithmes [LEE 98, LI 96] existent pour analyser les impacts du changement en analysant la structure du modèle étudié.

1.2.4. *Bilan*

La remodularisation de systèmes à objets reste un défi. Elle nécessite des outils variés comme des visualisations dédiées, l'analyse d'impact et les outils d'aide à la décision.

Les visualisations présentées se limitent à montrer l'architecture d'un système, elles ne permettent pas de mettre en évidence facilement les problèmes architecturaux. Les logiciels de gestion de version ne sont pas adaptés à la remodularisation car ils sont difficiles à manipuler pour comparer l'impact d'une nouvelle version sur la structure. Nous avons besoin d'un modèle de représentation du code source pouvant être manipulé et versionné tout en gardant une structure complète du système pour permettre l'analyse.

Dans les deux sections suivantes, une approche est proposée : l'analyse de la structure du logiciel est d'abord présentée grâce à une visualisation nommée eDSM basée sur *Dependency Structural Matrix*, ensuite un outil permettant de simuler les changements et d'aider le développeur à prendre les bonnes décisions pour la remodularisation de son système.

1.3. Identification et compréhension des dépendances entre paquetages

Nous présentons dans cette section une visualisation permettant au développeur de cibler les éléments à modifier pour remodulariser son système. Cette visualisation, basée sur la représentation d'un lien par une cellule, a l'avantage de montrer la structure globale de l'application au niveau des paquetages, et d'offrir dans le même contexte un aperçu des relations entre les classes et les méthodes en cause dans les problèmes de modularité.

La visualisation est basée sur le principe qu'une dépendance entre deux paquetages est définie par un ensemble de classes et de méthodes liées entre elles par quatre types de dépendance : l'héritage, l'invocation de méthode, la référence de classe et l'extension de classe. Ce principe est illustré dans la section 1.3.1 où une cellule de la matrice est présentée.

La construction de la visualisation se fait à l'aide de cellules. Elle a l'avantage d'être une visualisaton centrée dépendance plutôt que centrée paquetage. Elle permet donc de mettre en évidence les dépendances posant des problèmes de modularité. Les visualisations présentées ici aident à la résolution des cycles entre les paquetages. On

différencie les paquetages avec un lien unidirectionnel, les paquetages en cycle direct (deux paquetages communiquent entre eux dans les deux sens) et les paquetages en cycle complexe (un groupe de paquetages est en cycle).

1.3.1. *Un lien entre deux paquetages*

1.3.1.1. *Ajouter de l'information dans une cellule (case)*

Prenons une matrice de dépendances (comme celle de la figure 1.11). Dans notre cas, nous souhaitons connaître les liens entre les éléments internes aux paquetages pour comprendre la dépendance et pouvoir la briser si elle est indésirable. On donne donc dans chaque case une information détaillée des dépendances entre un paquetage source et un paquetage cible. Ainsi chaque case de la matrice représente un contexte isolé.

1.3.1.1.1. Aperçu d'une case

Une case est composée de quatre parties (figure 1.12) : un en-tête en haut donnant un aperçu du nombre et du type des liens dans cette dépendance. Les deux rectangles centraux représentent respectivement le paquetage source et le paquetage cible. On lit donc les dépendances du haut vers le bas. Le rectangle du bas représente l'état du lien dans le système.

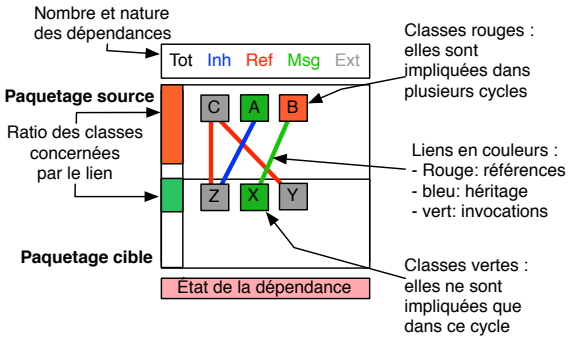

Figure 1.12 – Cellule représentant un lien entre deux paquetages

Les rectangles centraux, représentant chacun un paquetage sont composés des classes impliquées dans le cycle entre les deux paquetages, et seulement celles-ci. Les liens entre les classes sont colorés en fonction du type de dépendance entre les classes.

1.3.1.1.2. Aperçu du type de dépendance

La première information en haut d'une cellule est le nombre et le type des dépendances. Cinq nombres apparaissent : le nombre total de dépendances de cette cellule et le nombre de chaque type de dépendances (héritage, invocation de méthode, référence de classe, extension de classe[5]). A chaque type de dépendance correspond une couleur qui est ensuite utilisée pour tracer les liens entre les classes. L'héritage est en bleu, les références de classes en rouge, les invocations de méthodes en vert et les extensions de classes en gris.

1.3.1.1.3. Le contenu des paquetages

Le paquetage source et le paquetage cible sont représentés au centre de la cellule. Dans chacun d'eux, des informations quantitatives et qualitatives sont données pour aider le développeur à définir les actions à entreprendre. Trois informations sont affichées :

– Ratio des classes concernées. Pour chacun des paquetages, nous voulons connaître le nombre de classes concernées par rapport à la taille du paquetage. Cette barre est colorée (de rouge pour 100 % à vert pour 1 %) afin de renforcer l'impact visuel. De cette information dépendra une partie des décisions. Deux paquetages dont toutes les classes communiquent sont fortement corrélés. Dans le cas d'un cycle, on pourra se poser la question de les fusionner. Dans le cas où seulement très peu de classes sont concernées par la dépendance, on pourra se poser la question du déplacement de ces classes ;

– Affichage des classes concernées. Seules les classes concernées par la dépendance sont affichées. Elles sont colorées selon trois couleurs : rouge, vert ou gris. Une classe rouge signifie qu'elle est impliquée dans plusieurs cellules : effectuer des modifications sur cette classe peut avoir un impact important sur le reste du système : elle peut entraîner des effets indésirables ou au contraire, permettre la résolution de plusieurs cycles. Une classe verte, à l'inverse signifie que cette classe est utilisée uniquement dans les deux paquetages de la dépendance : ainsi ce type de classe peut être déplacé sans impact sur le reste du système. Les classes grises représentent toutes les autres classes, utilisées dans le reste du système sans générer de cycle.

– Des liens entre les classes. Les liens entre les classes sont de la couleur des dépendances énumérées dans l'entête de la cellule. Pour que l'affichage soit simplifié, ils apparaissent selon un ordre de priorité que l'on retrouve dans l'en-tête de la cellule : l'héritage est un lien plus fort que la référence de classe qui a plus d'importance que l'invocation. Si plusieurs liens coexistent, le lien est affiché en noir. L'extension de

5. Une extension de classe est une méthode définie dans un paquetage différent du paquetage de la classe [BER 05]. Ce type de construction existe en Smalltalk, CLOS, Ruby, Python, Objective-C et C-Sharp3. Il offre un moyen pratique pour modifier des classes lorsque le sous-classage est inapproprié.

classe est un cas particulier car la même classe est représentée dans les deux paquetages. Dans ce dernier cas, seule l'extension est affichée ;

– et les méthodes ? Les méthodes concernées par les dépendances (référence de classe et invocation de méthode) ne sont pas directement affichées. Pour obtenir cette information, une fenêtre furtive (*popup*) vient ajouter cette information sur chaque classe (figure 1.13).

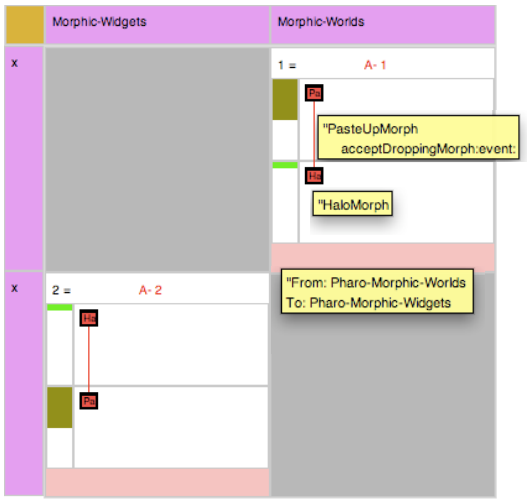

Figure 1.13 – Deux cellules représentant un cycle direct, les infobulles informent des méthodes concernées par le lien

1.3.1.1.4. Etat de la dépendance

La dernière information affichée dans cette visualisation donne l'état de la dépendance dans le système. Cet état dépend de la visualisation utilisée. Par exemple dans eDSM (section 1.3.2.1), il montre le type de cycle engendré (direct ou complexe). Alors que dans cycleTable (section 1.3.2.2), il représente le partage du lien entre les cycles.

1.3.2. *Mise en forme de ces informations*

Maintenant que l'on a modélisé un lien entre deux paquetages, nous pouvons créer des visualisations adaptées à la remodularisation d'un système à large échelle. Nous proposons ici deux visualisations : eDSM et CycleTable. La première est une visualisation permettant de comprendre la structure du système, elle ordonne les cellules dans

une matrice. La deuxième est une visualisation montrant les cycles. Elle les ordonne dans une table pour les comparer et mieux les résoudre.

1.3.2.1. *Visualisation dans une matrice : eDSM [LAV 09b]*

A l'origine, les matrices de dépendance sont utilisées pour identifier la succession de tâches dans l'optimisation de processus. C'est une technique reconnue pour son efficacité à détecter les cycles [SAN 05]. Elle représente une bonne alternative aux graphes pour visualiser la structure des logiciels [STE 81, SUL 01].

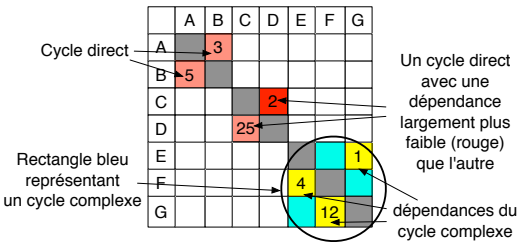

Figure 1.14 – Une DSM colorée

L'information contenue dans une cellule peut être un simple marqueur ou un poids comme dans la figure 1.10 ou toute autre information utile. Les paquetages (par exemple les lignes et les colonnes) sont ordonnancés en fonction du nombre de dépendances entrantes. Ainsi, un paquetage au cœur du système se trouve en bas et à droite de la matrice. Dans chacune des cellules représentant une dépendance, nous plaçons la visualisation vue précédemment et nous ajoutons l'information sur l'état de la cellule.

1.3.2.1.1. Etat d'une cellule : cycle direct ou complexe

L'état de la dépendance dans la matrice nous informe de sa modularité dans le système. Une dépendance entre deux paquetages peut être unidirectionnelle (couleur grise), c'est-à-dire qu'un paquetage dépend d'un autre et qu'il n'existe pas de chemin inverse. Deux paquetages peuvent être en cycle direct (rose/rouge), dans ce cas les deux paquetages communiquent entre eux dans les deux sens (figure 1.14, cycle {A,B} et {C,D}). Ils ne sont donc pas modulaires. Le dernier cas est le cas d'un cycle complexe (bleu) où plusieurs paquetages forment un cycle composé d'au moins trois dépendances (jaune) (figure 1.14, cycle {E,F,G}).

Rendre modulaire une application signifie que nous cherchons à avoir uniquement des liens unidirectionnels (gris). On concentrera nos efforts d'abord sur les cellules rouges/roses et ensuite sur les jaunes. Cet ordre est important car la suppression d'un cycle direct peut entraîner la suppression de cycles complexes.

L'objectif de cette visualisation étant d'aider le développeur, il nous semble important de lui donner des informations pour l'aider à corriger la structure. La visualisation ajoute donc une nuance rouge/rose pour montrer au développeur le lien le moins complexe (donc probablement le plus simple à supprimer) dans un cycle direct. Le cycle {C,D} de la figure 1.14 montre cette différenciation : en rouge la dépendance de C à D (poids 2) est moins « forte » que la dépendance de D à C (poids 25).

1.3.2.1.2. Supprimer un cycle grâce à eDSM.

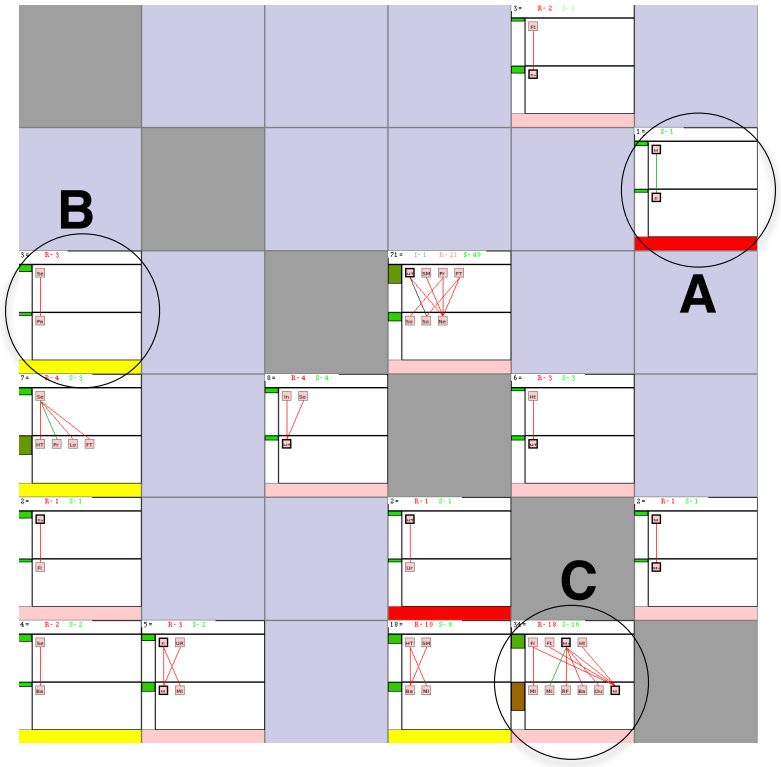

Figure 1.15 – Aperçu d'une eDSM : A – dépendance à supprimer, B – dépendance faisant partie d'un cycle non direct, C – cette dépendance a de nombreuses relations

Prenons l'exemple de la figure 1.15. Nous voyons très facilement (A) la case rouge représentant une dépendance à supprimer en premier lieu, (B) des cases jaunes représentant des dépendances à gérer après les cases rouges/roses, et (C) les cases avec de nombreuses relations entre classes. Dès le premier aperçu, nous voyons sur quelles dépendances travailler.

1.3.2.2. *Visualisation orienté cycle : cycleTable [LAV 09a]*

Nous venons de voir que eDSM permet d'identifier facilement les cycles directs et particulièrement les dépendances les moins complexes dans ces cycles. Cette visualisation est idéale pour commencer la remodularisation. Cependant, au bout de plusieurs itérations, il est probable qu'elle n'affiche plus de cycles directs.

La remodularisation devient plus difficile lorsque les dépendances restantes sont lourdes et que les cycles sont complexes. Pour cela, CycleTable permet de visualiser les cycles et met en valeur les dépendances partagées entre ces cycles. Cette méthode permet au développeur de concentrer ses efforts sur des dépendances stratégiques.

Une CycleTable est une matrice rectangulaire où chaque paquetage est représenté sur une ligne et chaque cycle est représenté par une colonne (figure 1.16 (droite)). CycleTable trace la séquence des dépendances dans un cycle et montre les liens partagés entre les cycles du système. La figure 1.16 (gauche) montre un graphe simple avec des cycles entrecroisés. Cette figure montre particulièrement qu'en brisant certains liens, on réussit à supprimer plusieurs cycles. Par exemple, supprimer le lien entre A et B permet de supprimer les deux cycles {A,B,C} et {A,B,E}.

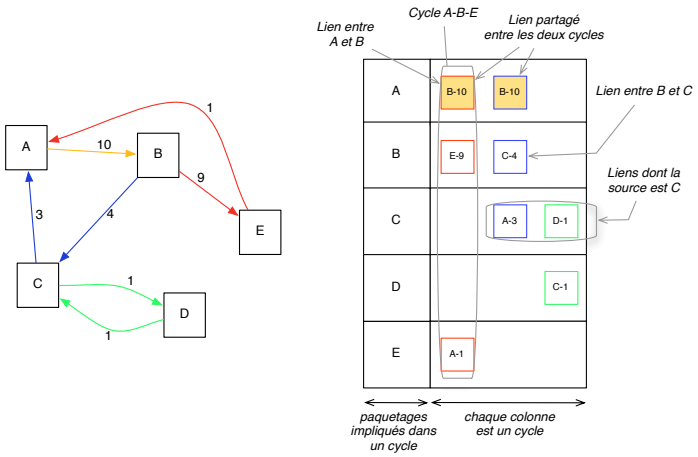

Figure 1.16 – Gauche : un graphe cyclique ; droite : structure de sa CycleTable

Cycle minimal

CycleTable montre uniquement les cycles minimaux. Un cycle minimal est un cycle qui ne passe pas deux fois par le même nœud, sauf pour le nœud de départ qui est également le nœud de fin. Par exemple, dans la figure 1.16, {A-B-E} et {A-B-C}

sont deux cycles minimaux, alors que {A-B-C-D-C} n'en est pas un car le nœud C apparaît deux fois.

Dépendance partagée

Une dépendance peut être partagée par plusieurs cycles, à l'image du lien entre A et B dans les cycles {A,B,C} et {A,B,E}. Ces dépendances sont mises en valeur dans cette visualisation par la partie « état de la dépendance » d'une cellule (voir section 1.3.1). Pour un même paquetage (sur une même ligne), si plusieurs cellules ont la même couleur cela signifie qu'elles représentent la même dépendance dans des cycles différents. En brisant cette dépendance, le développeur supprimera les cycles impliqués. Ainsi, le développeur peut porter son attention sur les couleurs les plus présentes.

Supprimer un cycle grâce à CycleTable

La figure 1.17 (droite) montre une CycleTable avec quatre paquetages du cœur de Pharo. En comparaison avec une eDSM (figure 1.17 (gauche)), nous apercevons plus de détails. Cinq cycles minimaux sont visibles et trois dépendances sont partagées (en rouge, bleu et orange). Le développeur peut donc concentrer ses efforts sur ces trois dépendances qui représentent des points stratégiques dans la remodularisation du système.

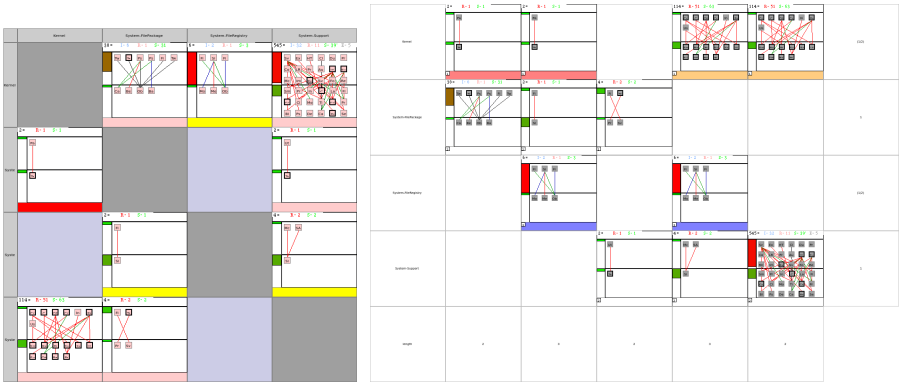

Figure 1.17 – Quatre paquetages de Pharo dans une eDSM (gauche) et dans une CycleTable (droite)

L'un des avantages de CycleTable est de ne pas fournir une seule proposition pour détruire un cycle, mais de présenter les différentes options possibles. Dans la figure 1.17 (droite), on constate que le lien rouge semble simple à briser, en revanche le choix entre le lien bleu et le lien orange est plus difficile : en effet, une fois le lien

rouge brisé, il reste deux cases oranges et une case bleue. Les cases oranges semblent être plus complexes que le lien bleu mais peuvent briser deux cycles. Dans ce cas c'est au développeur de faire le choix qu'il pense être le meilleur.

1.3.3. *Bilan des visualisations*

Nous avons présenté dans cette section une visualisation centrée sur les dépendances entre paquetages plutôt que sur les paquetages eux-mêmes. Cette visualisation structurée dans d'autres visualisations comme DSM ou CycleTable permet d'avoir une vision de l'architecture globale de l'application mais aussi d'obtenir des informations beaucoup plus détaillées sur la nature des dépendances.

Chacune des cellules ne montre qu'une seule dépendance d'un paquetage A vers un paquetage B, ce qui offre un contexte restreint et suffisant pour comprendre la qualité d'une dépendance et pouvoir se faire une idée de la remédiation à mettre en œuvre.

1.4. Simulation et aide à la prise de décision pour la remodularisation

Ces visualisations aident le développeur à trouver les sources des problèmes et, dans le meilleur des cas, le guident sur des pistes de solutions. Avec cette information, les choix du développeur sont faits selon son expérience et son intuition. Dans le monde de la remodularisation, nous souhaitons avoir des outils de simulation de changement qui permettent de comparer les impacts de différents changements.

L'outil présenté dans cette partie est un outil de simulation et de comparaison de changements. Nous présentons une approche permettant au développeur de simuler plusieurs futurs avec différents changements pour les comparer. Chaque futur est considéré comme un modèle à part entière, sur lequel nous pouvons utiliser les outils de réingénierie intégrés dans Moose, dont eDSM et CycleTable. A notre connaissance, il n'existe pas d'autres outils permettant de fournir une simulation sur modèle comme le permet celui-ci.

L'idée majeure est de simuler plusieurs futurs selon différents scénarios et les faire évoluer. Nous obtenons ainsi un arbre de versions simulées.

L'avantage de travailler sur des simulations est de ne pas rompre le flux de travail du développeur : effectuer des changements sur le code source, les versionner et les réimporter dans différents modèles pour les analyser et les comparer.

1.4.1. *Une vision de la réingénierie*

Prenons un scénario simple où le développeur doit retirer des dépendances cycliques entre paquetages. Notre expérience dans le domaine nous a régulièrement montré qu'il y a deux difficultés principales : choisir la bonne dépendance à supprimer et ne pas créer d'autres cycles en en enlevant une.

La première difficulté est en partie résolue grâce aux visualisations présentées précédemment. La deuxième est moins triviale, car éliminer une dépendance est souvent synonyme de restructuration (c'est à dire déplacement de classes, de méthodes, fusions, etc.).

Par exemple, la figure 1.18 présente trois paquetages, deux classes et deux méthodes *Model : :inferNamespaceParents* et *Model : :allNamespaces*[6]. Les dépendances entre ces paquetages créent deux cycles minimaux : {Moose-Core, Famix-Extensions, Famix-Core} et {Moose-Core, Famix-Core}. A partir d'une structure de base (a), nous avons plusieurs propositions de modifications (dont deux sont présentées en b et c). La structure modifiée en (b) enlève les deux cycles mais en crée un nouveau, alors que (c) supprime les deux cycles avec une seule action.

Conditions

Dans un cas comme celui que nous venons de présenter, il est difficile de connaître le résultat avant d'avoir implémenté la solution. Or le développeur choisira certainement une solution selon son intuition. La figure 1.18 montre précisément que deux actions similaires (déplacer une méthode) peuvent résoudre le problème. Cependant la meilleure solution n'est facilement identifiable qu'*a posteriori*, car déplacer la méthode change simultanément les relations entre plusieurs paquetages. Nous avons donc besoin d'outils pour comparer les propositions et valider la meilleure. Nous énonçons les conditions à réunir pour réussir cette étape.

> [Req1] Le développeur a besoin d'outils d'analyse pour valider la situation d'une simulation. Il adaptera ces outils en fonction du besoin : métriques, visualisations, algorithmes. Une librairie d'outils de réingénierie est grandement utile.
>
> [Req2] Les actions de remodularisation doivent avoir une granularité adaptée à la tâche effectuée. Par exemple, la fusion de deux classes implique des modifications des méthodes, cela doit être fait automatiquement.
>
> [Req3] Les outils doivent fonctionner sur n'importe quelle version simulée. Souvent les outils de versionnage stockent uniquement les changements entre

6. Notons que inferNamespaceParents et allNamespaces sont définies dans des paquetages différents de celui de leur classe. Cette fonctionnalité appelée extension de classe est particulièrement utile pour rendre les paquetages plus modulaires.

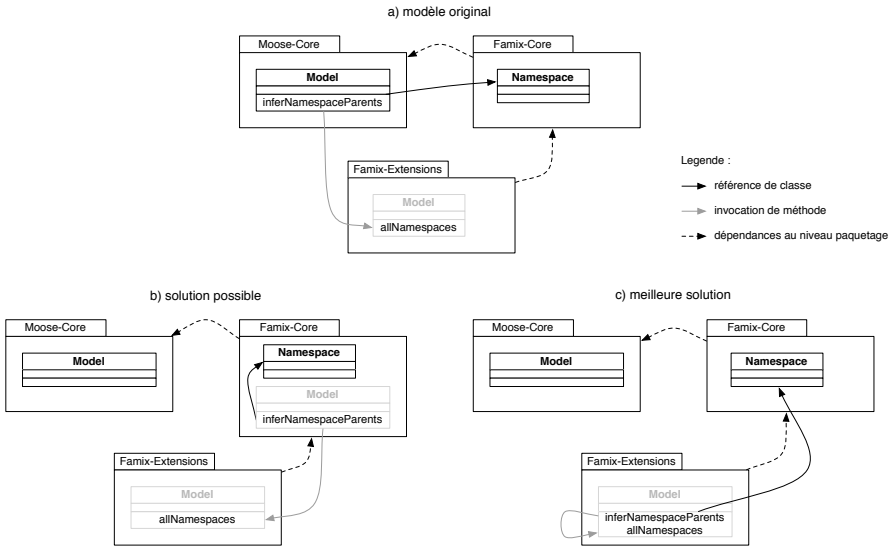

Figure 1.18 – a) deux cycles entre trois paquetages ; b) un changement qui enlève les deux cycles mais en crée un nouveau ; c) un changement qui enlève les deux cycles.

les versions. Cela ne permet pas aux outils de fonctionner sur une version comme sur un modèle complet. Chaque version doit avoir une vision complète du système.

[Req4] Le développeur doit pouvoir naviguer entre les différentes versions et ajouter une nouvelle version quand il veut et où il veut. L'ingénieur doit pouvoir sélectionner une version simplement et dériver cette version pour apporter des modifications. Il doit également pouvoir s'arrêter sur une version et la désigner comme une version majeure.

[Req5] Le développeur a besoin d'outils de comparaison de versions pour analyser l'impact des changements. Par exemple, dans le cadre de la réduction de cycles, le développeur doit pouvoir analyser l'état des cycles après une action, vérifier si les cycles ciblés ont bien été supprimés, si de nouveaux cycles ne sont pas apparus.

[Req6] Le développeur a besoin d'un outil lui permettant d'adapter les changements dans son code source. Une fois qu'une version est validée, l'ingénieur a besoin d'appliquer la séquence d'actions à réaliser dans son code source pour atteindre le résultat obtenu en simulation.

Pour répondre aux conditions *[Req1, Req2, Req5]*, des outils appropriés aux simulations et aux modifications de modèles sont nécessaires. Nous considérons ces outils comme spécifiques à chaque type de tâche. Alors que pour répondre aux conditions *[Req3, Req4, Req6]*, une architecture adaptée aux outils déjà existants doit être mise en place. Cette architecture doit avoir une certaine forme de généricité.

1.4.2. *Difficultés de navigation*

Les modèles utilisés dans la réingénierie en général sont importants en taille car ils réifient de nombreux éléments pour permettre des analyses fines. Par exemple, le modèle de Pharo, une plate-forme Smalltalk *open source* contient 1 800 classes contenues dans 150 paquetages. Sa représentation dans un modèle FAMIX contient plus de 800 000 entités, car FAMIX crée des entités pour les variables, les accès, les invocations, etc.

Dans une approche où plusieurs versions sont accessibles, la question de la gestion des modèles se pose. L'approche la plus intuitive est de créer un modèle complet pour chaque version, ce qui signifie dans le cadre de l'exemple de Pharo de copier les 800 000 entités. Ce processus prend du temps pour la copie et de la place en mémoire. Une autre idée est d'utiliser la copie partielle : elle consiste à copier uniquement les entités modifiées et toutes les entités en relation avec celles-ci. Cette approche n'est pas fonctionnelle sur des modèles de logiciels de grande taille car généralement l'ensemble des entités sont en relation. Il en résulte une copie complète du modèle par transitivité.

Notre solution est une simplification de la copie partielle : on copie uniquement les entités modifiées et leur conteneur (un paquetage pour une classe, une classe pour une méthode, etc.) et on partage le reste des entités. La figure 1.19 montre comment une table commune des entités partagées est utilisée pour minimiser la copie. En couleur cyan sont représentées les entités partagées, les deux grises et la croix sont respectivement la création, la modification et la suppression d'entités. Ces modifications n'ont aucun impact sur les versions précédentes du système. Cependant, comme nous le montrons dans [LAV 10], ce mécanisme doit être beaucoup plus subtile qu'il n'y paraît à cause de la navigation entre éléments modifiés dans différentes versions. Une modification faite dans une version récente doit prendre la précédence sur les versions ultérieures et ceci même si les élements sont accédés par des références émanant d'éléments d'une version ultérieure.

Ce système de partage requiert une gestion particulière des entités, mais il offre un compromis entre la gestion de la mémoire et l'accès aux entités de chaque version. De plus, le fait d'enregistrer uniquement les éléments modifiés permet de connaître l'évolution de l'architecture.

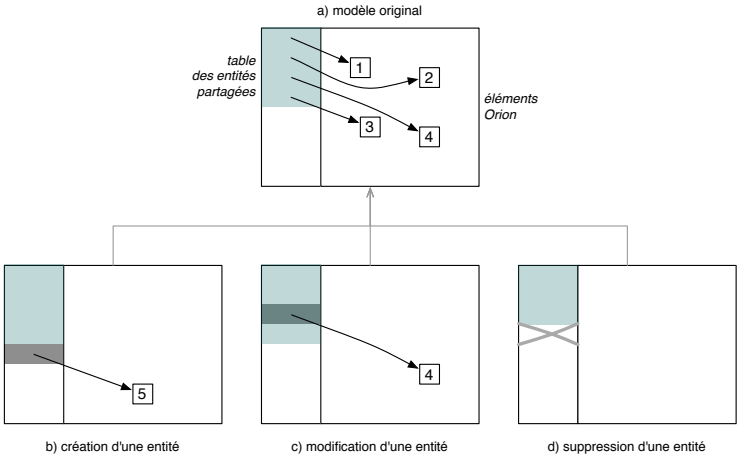

Figure 1.19 – Le système de partage d'Orion : illustration des différents changements (création, modification, suppression)

1.4.3. *L'utilisation d'Orion pour la réingénierie : un navigateur adaptable*

Orion est une approche mettant en pratique les six conditions énoncées précédemment. Un navigateur de versions a été créé et permet de réunir l'analyse de structures et la simulation de changements dans un même outil.

La figure 1.20 montre le navigateur composé de deux parties principales : en haut se trouve la navigation dans les versions. Il répond aux conditions [Req2, Req4, Req6]. En bas se trouvent les outils d'analyse pouvant être adaptés selon le besoin. Ici nous montrons des outils utiles pour la remodularisation de paquetages. Cette partie répond aux conditions [Req1, Req3, Req5]. Il est à noter que ce navigateur n'est pas une fin en soi mais simplement une illustration des possibilités de navigation et dévaluation entre versions d'un même logiciel.

1.4.3.1. *Navigation dans un modèle*

La partie haute du navigateur permet la navigation dans les versions : à gauche l'arbre de versions ; à droite les deux panneaux permettent de naviguer dans une version sélectionnée dans le panneau précédent. Ce panneau montre d'abord des groupes d'entités (tous les paquetages, toutes les classes etc.), il est possible de voir les entités en sélectionnant un groupe. Ensuite, en sélectionnant une entité, un nouveau panneau à droite affiche l'ensemble des entités en relation avec celle sélectionnée (dans la figure 1.20, le panneau de droite montre toutes les entités (méthodes, classes, paramètres, etc.) en relation avec la méthode sélectionnée dans le panneau du milieu).

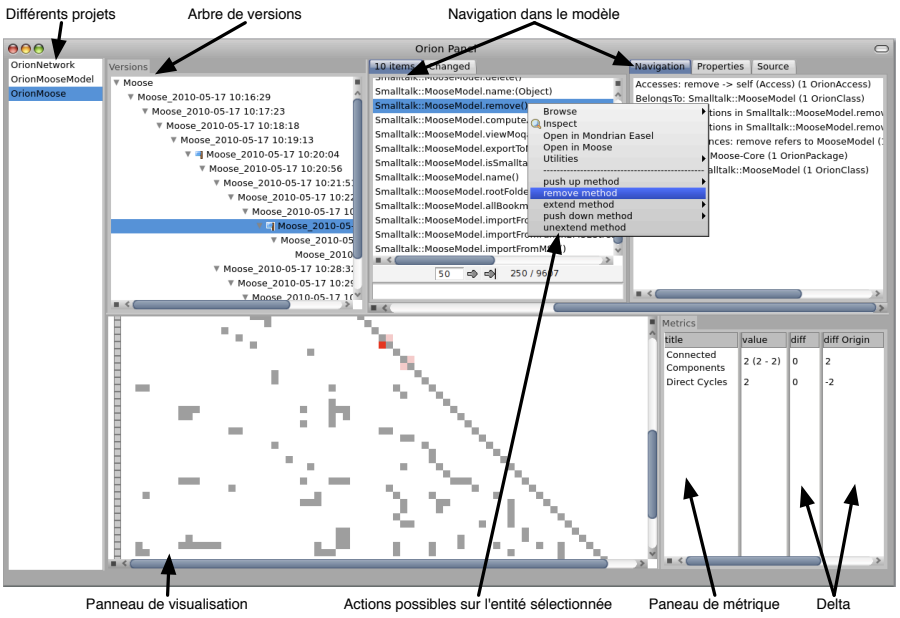

Figure 1.20 – Un navigateur Orion, dédié à la suppression des cycles entre paquetages

Dans l'arbre des versions, il est possible de créer une nouvelle version à partir d'une version sélectionnée, de supprimer une version ou de voir la séquence des actions réalisées jusqu'à la version sélectionnée. Ces actions sont accessibles par le menu contextuel.

Chaque entité dans le second panneau a une liste d'opérations de réingénierie possibles. Par exemple, dans la figure 1.20, la méthode sélectionnée peut être étendue dans un autre paquetage (*extend method*), supprimée (*remove method*), poussée dans une super-classe ou une sous-classe (*push up/down method*).

Appliquer une action génère un changement dans la version en cours d'édition sauf si elle a des versions enfants. Dans ce cas, pour conserver la consistance, une nouvelle version est créée.

1.4.3.2. *Analyses du modèle*

La partie basse du navigateur est dédiée à l'analyse et à la validation des versions. Elle contient deux panneaux : l'un pour une visualisation, l'autre pour des métriques. Cette partie est adaptable selon les besoins du développeur, à l'aide des outils disponibles dans l'environnement.

Dans la figure 1.20, le panneau gauche montre la visualisation de DSM appliquée à une version. Il est possible en sélectionnant un cycle particulier d'afficher la visualisation eDSM pour voir les détails des cycles. Il est possible de mettre d'autres visualisations dans ce cadre mais nous conseillons une visualisation simple car il doit représenter uniquement un point de départ de l'analyse. A partir de ce point, il est possible d'utiliser des outils d'analyse plus complexes et plus spécifiques.

La partie droite du panneau analyse du navigateur est destinée aux métriques. Ici nous affichons deux métriques (nombre de composantes connexes et nombre de cycles directs) se déclinant en trois valeurs : la valeur calculée dans la version en cours d'étude, la différence avec la version précédente et la différence avec la version d'origine. Ces métriques servent à mesurer la progression vers l'objectif principal (ici la suppression des cycles entre paquetages).

1.4.4. *Bilan*

L'idée de simuler plusieurs futurs selon différents scénarios et de les faire évoluer permet de mieux décider des actions à entreprendre lors de la remodularisation d'une application. Ainsi, les choix du développeur peuvent être testés sans conséquence sur le code source et validés selon leurs impacts.

L'outil présenté dans cette partie est un outil de simulation et de comparaison de changements. Nous avons présenté un modèle Orion permettant de naviguer et de modifier des modèles tout en restant intégrés avec les outils de réingénierie existants. Ainsi, Orion permet à l'aide des visualisations présentées précédemment d'établir un plan de remodularisation, à réaliser ensuite sur le code source.

1.5. Conclusion

Ce chapitre était dédié à la manipulation de systèmes dans l'optique de les rendre modulaires. Nous avons montré dans un premier temps que les outils couramment utilisés s'adaptent difficilement aux travaux de remodularisation.

D'un côté, les outils de visualisation montrent bien souvent la structure du système sans mettre en évidence les problèmes que les développeurs aimeraient soulever. Nous avons présenté une idée de visualisation permettant de remédier ce problème en utilisant l'idée de petits multiples de Tufte (*small multiples*) et de la coloration des structures.

D'un autre côté, les logiciels de gestion de versions ne sont pas adaptés à la navigation et à la manipulation que l'on souhaite faire lors de la remodularisation. Nous

avons donc présenté une approche nommée Orion permettant de manipuler des versions d'un programme, visualiser la structure des versions, voir les impacts des changements entre les versions, dans un même outil et surtout dans un même flot de travail.

Les systèmes communément utilisés sont limités pour la remodularisation et un défi d'aujourd'hui est de créer les approches et les outils pour rendre plus abordable la remodularisation.

1.6. Bibliographie

[ABD 08] ABDEEN H., ALLOUI I., DUCASSE S., POLLET D., SUEN M., « Package Reference Fingerprint : a Rich and Compact Visualization to Understand Package Relationships », *European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society Press, p. 213–222, 2008.

[ABD 09a] ABDEEN H., Visualizing, Assessing and Re-Modularizing Object-Oriented Architectural Elements, PhD thesis, Université de Lille, 2009.

[ABD 09b] ABDEEN H., DUCASSE S., SAHRAOUI H. A., ALLOUI I., « Automatic Package Coupling and Cycle Minimization », *International Working Conference on Reverse Engineering (WCRE)*, Washington, DC, Etats-Unis, IEEE Computer Society Press, p. 103–112, 2009.

[ABD 10] ABDEEN H., DUCASSE S., POLLET D., ALLOUI I., « Package Fingerprint : a visual summary of package interfaces and relationships », *Information and Software Technology Journal*, vol. 52, p. 1312-1330, 2010.

[ABD 11] ABDEEN H., DUCASSE S., SAHRAOUI H. A., « Modularization Metrics : Assessing Package Organization in Legacy Large Object-Oriented Software (short paper) », *International Working Conference on Reverse Engineering (WCRE'11)*, Washington, DC, Etats-Unis, IEEE Computer Society Press, p. ?– ?, 2011.

[ARI 04] ARISHOLM E., BRIAND L. C., FOYEN A., « Dynamic Coupling Measurement for Object-Oriented Software », *IEEE Transactions on Software Engineering*, vol. 30, n°8, p. 491–506, 2004.

[BAL 05] BALZER M., DEUSSEN O., LEWERENTZ C., « Voronoi treemaps for the visualization of software metrics », *SoftVis '05 : Proceedings of the 2005 ACM symposium on Software visualization*, New York, NY, Etats-Unis, ACM, p. 165–172, 2005.

[BAR 01] BARLOW T., NEVILLE P., « A comparison of 2-D Visulization of Hierarchies », *Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, 2001.

[BER 83] BERTIN J., *Semiology of Graphics*, University of Wisconsin Press, 1983.

[BER 05] BERGEL A., DUCASSE S., NIERSTRASZ O., « Analyzing Module Diversity », *Journal of Universal Computer Science*, vol. 11, n°10, p. 1613–1644, novembre 2005.

[BOH 96] BOHNER S. A., ARNOLD R. S., *Software Change Impact Analysis*, IEEE Computer Society Press, 1996.

[BRI 99] BRIAND L. C., DALY J. W., WÜST J. K., « A Unified Framework for Coupling Measurement in Object-Oriented Systems », *IEEE Transactions on Software Engineering*, vol. 25, n°1, p. 91–121, 1999.

[CHA 02] CHAUMUN M. A., KABAILI H., KELLER R. K., LUSTMAN F., « A change impact model for changeability assessment in object-oriented software systems », *Science of Computer Programming*, vol. 45, n°2-3, p. 155 - 174, 2002.

[D'A 06] D'AMBROS M., LANZA M., « Reverse Engineering with Logical Coupling », *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, p. 189 - 198, 2006.

[DEM 99] DEMEYER S., DUCASSE S., LANZA M., « A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization », BALMAS F., BLAHA M., RUGABER S., Eds., *Proceedings of 6th Working Conference on Reverse Engineering (WCRE '99)*, IEEE Computer Society, octobre 1999.

[DEM 01] DEMEYER S., TICHELAAR S., DUCASSE S., FAMIX 2.1 — The FAMOOS Information Exchange Model, Rapport, University of Bern, 2001.

[DEM 02] DEMEYER S., DUCASSE S., NIERSTRASZ O., *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2002.

[DUC 06] DUCASSE S., GÎRBA T., KUHN A., « Distribution Map », *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, Los Alamitos CA, IEEE Computer Society, p. 203–212, 2006.

[DUC 07] DUCASSE S., POLLET D., SUEN M., ABDEEN H., ALLOUI I., « Package Surface Blueprints : Visually Supporting the Understanding of Package Relationships », *ICSM '07 : Proceedings of the IEEE International Conference on Software Maintenance*, p. 94–103, 2007.

[DUC 09a] DUCASSE S., GÎRBA T., KUHN A., RENGGLI L., « Meta-Environment and Executable Meta-Language using Smalltalk : an Experience Report », *Journal of Software and Systems Modeling (SOSYM)*, vol. 8, n°1, p. 5–19, Springer Verlag, février 2009.

[DUC 09b] DUCASSE S., POLLET D., « Software Architecture Reconstruction : A Process-Oriented Taxonomy », *IEEE Transactions on Software Engineering*, vol. 35, n°4, p. 573-591, juillet 2009.

[Gˆ 05a] GÎRBA T., KUHN A., SEEBERGER M., DUCASSE S., « How Developers Drive Software Evolution », *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, IEEE Computer Society Press, p. 113–122, 2005.

[Gˆ 05b] GÎRBA T., LANZA M., DUCASSE S., « Characterizing the Evolution of Class Hierarchies », *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, Los Alamitos CA, IEEE Computer Society, p. 2–11, 2005.

[HAN 97] HAN J., « Supporting Impact Analysis and Change Propagation in Software Engineering Environments », *STEP '97 : Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP '97) (including CASE '97)*, Washington, DC, Etats-Unis, IEEE Computer Society, page172, 1997.

[HAU 02] HAUTUS E., « Improving Java Software through Package Structure Analysis », *IASTED International Conference Software Engineering and Applications*, 2002.

[HEN 07] HENRY N., FEKETE J.-D., MCGUFFIN M. J., « NodeTrix : a Hybrid Visualization of Social Networks », *IEEE Trans. Vis. Comput. Graph.*, vol. 13, n°6, p. 1302-1309, 2007.

[JOH 91] JOHNSON B., SHNEIDERMAN B., « Tree-Maps : a space-filling approach to the visualization of hierarchical information structures », *VIS '91 : Proceedings of the 2nd conference on Visualization '91*, Los Alamitos, CA, Etats-Unis, IEEE Computer Society Press, p. 284–291, 1991.

[KAR 03] KAROUACH S., Système de visualisations interactives pour la découverte de connaissances, PhD thesis, Université Paul Sabatier, Toulouse III, juillet 2003.

[LAN 03a] LANZA M., Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization, PhD thesis, University of Bern, mai 2003.

[LAN 03b] LANZA M., DUCASSE S., « Polymetric Views—A Lightweight Visual Approach to Reverse Engineering », *Transactions on Software Engineering (TSE)*, vol. 29, n°9, p. 782–795, IEEE Computer Society, septembre 2003.

[LAN 05] LANGELIER G., SAHRAOUI H., POULIN P., « Visualization-based analysis of quality for large-scale software systems », *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, New York, NY, Etats-Unis, ACM, p. 214–223, 2005.

[LAV 09a] LAVAL J., DENIER S., DUCASSE S., « Identifying cycle causes with CycleTable », *FAMOOSr 2009 : 3rd Workshop on FAMIX and MOOSE in Software Reengineering*, Brest, France, 2009.

[LAV 09b] LAVAL J., DENIER S., DUCASSE S., BERGEL A., « Identifying cycle causes with Enriched Dependency Structural Matrix », *WCRE '09 : Proceedings of the 2009 16th Working Conference on Reverse Engineering*, Lille, France, 2009.

[LAV 10] LAVAL J., DENIER S., DUCASSE S., FALLERI J.-R., « Supporting Simultaneous Versions for Software Evolution Assessment », *Journal of Science of Computer Programming (SCP)*, mai 2010.

[LEE 98] LEE M. L., Change impact analysis of object-oriented software, PhD thesis, George Mason University, Fairfax, VA, Etats-Unis, 1998, Director-Jeff Offutt.

[LI 96] LI L., OFFUTT A. J., « Algorithmic Analysis of the Impact of Changes to Object-Oriented Software », *ICSM '96 : Proceedings of the 1996 International Conference on Software Maintenance*, Washington, DC, Etats-Unis, IEEE Computer Society, p. 171–184, 1996.

[MAR 03] MARTIN R. C., *Agile Software Development : principles, patterns and practices*, Prentice-Hall, 2003.

[NGU 05] NGUYEN T., MUNSON E., BOYLAND J., « An Infrastructure for Development of Object-Oriented, Multi-level Configuration Management Services », *Internationl Conference on Software Engineering (ICSE 2005)*, ACM Press, p. 215–224, 2005.

[NIE 05] NIERSTRASZ O., DUCASSE S., GÎRBA T., « The Story of Moose : an Agile Reengineering Environment », *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, New York NY, ACM Press, p. 1–10, 2005, Invited paper.

[PAR 72] PARNAS D. L., « On the Criteria To Be Used in Decomposing Systems into Modules », *CACM*, vol. 15, n°12, p. 1053–1058, décembre 1972.

[POL 07] POLLET D., DUCASSE S., POYET L., ALLOUI I., CÎMPAN S., VERJUS H., « Towards A Process-Oriented Software Architecture Reconstruction Taxonomy », KRIKHAAR R., VERHOEF C., DI LUCCA G., Eds., *Proceedings of 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, IEEE Computer Society, mars 2007, Best Paper Award.

[SAN 05] SANGAL N., JORDAN E., SINHA V., JACKSON D., « Using Dependency Models to Manage Complex Software Architecture », *Proceedings of OOPSLA'05*, p. 167–176, 2005.

[STE 81] STEWARD D., « The design structure matrix : A method for managing the design of complex systems », *IEEE Transactions on Engineering Management*, vol. 28, n°3, p. 71–74, 1981.

[SUL 01] SULLIVAN K. J., GRISWOLD W. G., CAI Y., HALLEN B., « The Structure and Value of Modularity in SOftware Design », *ESEC/FSE 2001*, 2001.

[TUF 01] TUFTE E. R., *The Visual Display of Quantitative Information*, Graphics Press, 2nd édition, 2001.

[WAR 00] WARE C., *Information visualization : perception for design*, Morgan Kaufmann Publishers Inc., San Francisco, CA, Etats-Unis, 2000.

[WET 07] WETTEL R., LANZA M., « Program Comprehension through Software Habitability », *Proceedings of ICPC 2007 (15th International Conference on Program Comprehension)*, IEEE CS Press, p. 231–240, 2007.

[WET 08] WETTEL R., LANZA M., « Visual Exploration of Large-Scale System Evolution », *In Proceedings of WCRE 2008 (15th IEEE Working Conference on Reverse Engineering)*, IEEE CS Press, p. 219 - 228, 2008.